Chapter – 2

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**Perceptrons**

Lectures: Geoffrey Hinton

Types of neural network architectures

Perceptrons: first generation neural networks

A geometrical view of perceptrons

Why the learning works

What perceptrons can't do

**Chapter 2: Perceptrons**

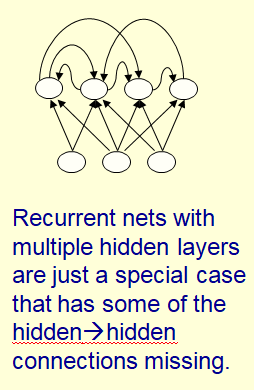
**2.1 Types of neural network architectures**

In this section we'll discuss about the architectures of different neural networks. The architecture means the way the neurons are connected together.

* Feed forward NN: Feed forward neural network is a common type of architecture. In Feed-Forward NN, the information comes into the input units and flows in one direction through hidden layers until each reaches the output units.
* RNN: A much more interesting kind architecture is a recurrent neural network (RNN) in which information can flow round in cycles. These networks can remember information for a long time.
* They can exhibit all sorts of interesting oscillations.
* They are much more difficult to train because they are so much more complicated.
* Symmetrically-Connected NN: In a symmetrically-connected network the weights are the same in both directions between two units.

|  |  |
| --- | --- |
| * Feed-Forward NN: The most common type of neural network in practical applications is a Feed-Forward Neural Network. * This has some input units (figure: the first layer at the bottom), some output units (figure: the last layer at the top), * Also, there are one or more layers of hidden units (figure: middle horizontal layer). * If there's more than one layer of hidden units, we call them Deep Neural Networks. |  |

* These networks compute a series of transformations between their input and their output.
* So at each layer, you get a new representation of the input in which things that were *similar in the previous layer* may have become *less* *similar*, or things that were *dissimilar in the previous layer* may have become *more similar*.
* e.g. in *speech recognition*, we'd like the *same thing said by different speakers* to become *more similar*, and *different things* said by the *same speaker* to be *less similar* as we *go up through the layers* of the *network*.
* In order to achieve this, we need the activities of the neurons in each layer to be a **non-linear function** of the *activities* in the *layer below*.
* Recurrent Neural Networks: RNNs are much more powerful than Feed Forward Neural networks.
* They have directed cycles in their connection graph.
* It means: If you start at a node or a neuron and you follow the arrows, you can sometimes get back to the neuron you started at.
* They can have very complicated dynamics, and this can make them very difficult to train.
* There is a lot of interest at present in finding efficient ways of training the RNNs. Because they are so powerful if we can train them.
* They are more biologically realistic.
* **RNNs** with multiple hidden layers are just a special case of a **general RNN** that has some of it's hidden to hidden connections missing.



|  |  |
| --- | --- |
| * RNN & Sequence: RNNs are a very natural way to model sequential data. * We have connections between hidden units. And the hidden units act like a network that's very deep in time. * They are equivalent to very deep nets with one hidden layer per time slice. * At *each time step* the states of the *hidden units* determines the *states* of the *hidden units* of the next time step. * Except that they use the *same weights* at *every time slice* and they get *input* at *every time slice*. * Unlike feed-forward nets in RNN we used the same weights at every time step. * So if you look at those red arrows where the hidden units are determining the next state of the hidden units, the weight matrix depicted by each red arrow is the same at each time step. * They also get inputs at every time step and often give outputs at every time step, and those will use the same weight matrices too. * RNNs have the ability to remember information in the hidden state for a long time. But it's very hard to train them to use this potential. |  |

|  |  |
| --- | --- |
| * Example of RNNs ability: An example of what recurrent neural nets can now do. Ilya Sutskever (2011) trained a special type of RNN to predict the next character in a sequence. * After training for a long time on a *string* of *half a billion characters* from *English Wikipedia*, he got it to generate new text. * It generates by predicting the probability distribution for the next character and then sampling a character from this distribution. * If you give it a string of characters and get it to predict probabilities for the next character. |  |

* Notice some sentences makes no sense on specific topics but they are grammatically not wrong. It's almost good English.
* Notice also the thing it says at the end, "such that it is the blurring of appearing on any well-paid type of box printer" , it's context is now about appearance and printing, and the syntax is pretty good.
* Symmetrically Connected Networks: These are like RNNs, but the connections between units are symmetrical (they have the same weight in both directions).
* John Hopfield (and others) realized that symmetric networks are much easier to analyze than RNN.
* They are much easier to analyze because they are more restricted in what they can do because they obey an energy function.
* For example, they cannot model cycles. You can't get back to where you started in one of these symmetric networks.
* Symmetrically connected nets without hidden units are called “Hopfield nets”.
* Symmetrically connected networks with hidden units (BM): These are called “Boltzmann machines”.
* They are much more powerful models than Hopfield nets.
* They are less powerful than RNNs.
* They have a beautifully simple learning algorithm.

**2.2 PERCEPTRONS: first generation neural networks**

The PERCEPTRONS were investigated in the early 1960's, and initially they looked very promising as learning devices. But then they fell into **disfavor** because **Minsky** and **Papert** showed they were rather restricted in what they could learn to do.

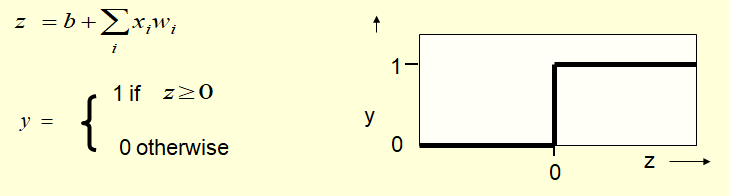
* Statistical Pattern Recognition System: In *statistical pattern recognition*, there's a statistical way to recognize patterns.
* We first take the raw input, and we convert it into a set or vector of feature activations.
* We do this using hand written programs which are based on common sense (that part of the system does not learn).
* We look at the problem we decide what the good features should be.
* We try some features to see if they work or don't work.
* We try some more features and eventually set of features that allow us to solve the problem by using a subsequent learning stage.
* What we learn is: How to weight each of the feature activations, in order to get a single scalar quantity.
* So the weights on the features represent how much evidence the feature gives you, in favor or against the hypothesis that the *current* *input* is an example of the kind of *pattern* you want to *recognize*.
* When we add up all the weighted features, we get a sort of total evidence in favor of the hypothesis that *this is the kind of pattern we want to recognize*.
* If that evidence is above some threshold, we decide that the input vector is a positive example of the class of patterns we're trying to recognize.
* The standard paradigm for statistical pattern recognition:
* Convert the raw input vector into a vector of feature activations.
* Use hand-written programs based on common-sense to define the features.
* Learn how to weight each of the feature activations to get a *single scalar quantity*.
* If this quantity is above some threshold, decide that the input vector is a positive example of the target class.

|  |  |
| --- | --- |
| * Perceptron: A PERCEPTRON is a particular example of a *statistical pattern recognition system*. * There are actually many different kinds of perceptrons, but the standard kind, which Rosenblatt called an alpha perceptron, consists of some inputs which are then converted into feature activities. * They might be converted by things that look a bit like neurons, but that stage of the system does not learn. * Once you've got the activities of the features, you then learn some weights, * so that you can take the . * Then you decide whether or not is an example of the class you're interested in by seeing whether that sum of is greater than a threshold. |  |

* History of Perceptrons: Perceptrons were popularized in the early 1960s by Frank Rosenblatt. He wrote a great big book called Principles of Neurodynamics, in which he described many different kinds of perceptrons, and that book was full of ideas.
* The most important thing in the book was a very Powerful Learning Algorithm.
* A lot of grand claims were made for what perceptrons could do using this learning algorithm.
* For example, people claimed that Perceptrons could tell the difference between pictures of tanks and pictures of trucks, even if the tanks and trucks were sort of partially obscured in a forest.
* But some of those claims turned out to be false. In the case of the tanks and the trucks, it turned out the pictures of the tanks were taken on a sunny day, and the pictures of the trucks were taken on a cloudy day.
* All the perceptron was doing was measuring the total intensity of all the pixels.
* But a human can notice the things in the picture.
* In 1969, Minsky and Papert published a book called Perceptrons that analyzed a perceptrons ability & limitations.
* Many people thought those limitations applied to all neural network models too.
* But Minsky and Papert just shown that *perceptrons* of the kind for which the *powerful learning algorithm applied* could not do a lot of things.
* They could do some work, only if you sort of hand-written the answer in the inputs, but not by learning.
* Some people in artificial intelligence also de-motivated *G. Hinton* during his work on neural network models in the 1970s, by telling him that "***Minsky*** and ***Papert*** have proved that these ***models were no good***".
* Actually, the perceptron convergence procedure, is still widely used today for tasks that have very big feature vectors.

The perceptron learning procedure is still widely used today for tasks with enormous feature vectors that contain many millions of features.

* Binary threshold neurons (decision units): The decision unit in a perceptron is a binary threshold neuron.
* They compute a weighted sum of inputs they get from other neurons.
* They add on a bias to get their total input.
* Then they give an output of **1** if that sum exceeds zero, and they give an output of **0** otherwise.



|  |  |
| --- | --- |
| * Learn Biases: How to learn biases using the same rule as we use for learning weights ? * Biases: We don't want to have to have a separate learning rule for learning biases, and it turns out we can treat biases just like weights. * If we take every input vector and we stick a **1** on the ***front of it***, and we treat the bias as like the weight on that first feature that always has a ***value*** of ***1***. * A threshold is equivalent to having a negative bias. * A bias is exactly **equivalent** to a **weight** on an **extra input line** that always has an **activity** of **1**. * And using this trick, we don't need a separate learning rule for the bias. We can now learn a bias as if it were a weight. |  |

* The Perceptron Convergence Procedure: (Training binary output neurons as classifiers) It's a very powerful learning procedure for PERCEPTRONS, and it's a learning procedure that's guaranteed to work.
* Add an extra component with value 1 to each input vector. The “bias” weight on this component is minus the threshold. Now we can forget the threshold (bias).
* So we first had this extra component with a value of **1** to every input vector. Now we can forget about the biases.
* We *keep picking training cases*, using any policy we like, as long as we ensure that *every training case gets picked without waiting too long*. Pick training cases using any policy that ensures that every training case will keep getting picked. Now, having picked a training case, you look to see if the output's correct.
* If the output unit is correct, leave its weights alone.
* If the output unit incorrectly outputs a ***zero***, ***add*** the input vector to the weight vector.
* I.e. if the output unit outputs a zero when it should've output a one, in other words, it said it's not an instance of the pattern we're trying to recognize, when it really is.
* Then all we do is we add the input vector to the weight vector of the perceptron.
* If the output unit incorrectly outputs a ***1*** ( when is should have output a zero), ***subtract*** the input vector from the weight vector of the perceptron.
* This is guaranteed to find a set of weights that gets the right answer for all the training cases if any such set exists.
* Means that it can only find the set of weight if the weights actually exists. But for many interesting problems there is no such set of weights.
* The existence of such set of weights depends very much on what features you use.
* It turns out for many problems the difficulty is deciding what features to use.
* If you're using the appropriate features learning then the training may become easy.
* If you're not using the right features, learning becomes impossible. So the most important work is deciding the features.

**2.3 A geometrical view of PERCEPTRONS**

In this section, we're going to get a geometrical understanding of what happens when a perceptron learns.

* Weight space: It's a high dimensional space in which each point corresponds to a particular setting for all the weights.
* In this space, we can represent the training cases as planes and learning consists of trying to get the weight vector on the right side of all the training planes.
* One dimension per weight: This is the space that has one dimension for each weight in the perceptron.
* A point in the space represents a particular setting of all the weights.
* Assuming we've eliminated the threshold, we can represent every *training case* as a *hyperplane* through the *origin* in weight space.
* The *weights* must *lie* on one side of this *hyper-plane* to get the *answer correct*.
* So, points in the space correspond to weight vectors and training cases correspond to planes.
* And, for a particular training case, the weights must lie on one side of that hyperplane, in order to get the *answer correct* for that training case.
* Let's look at a picture: So, let's look at a geometric picture of weight space so that we can understand what's going on.
* Each ***training case*** defines a ***plane*** (shown as a black line)
* The ***plane*** goes through the ***origin*** and is ***perpendicular*** to the ***input*** vector.
* On *one side* of the plane the output is *wrong* because the *scalar product* of the *weight* *vector* with the *input* *vector* has the *wrong* *sign*.
* Training Case: Let's think of one training case for now, it defines a plane, but in this 2D picture it is just the black line.
* Input Vector: The plane goes through the origin and it's perpendicular to the input vector for that training case, which here is shown as a *blue vector*.
* Case 1 (correct is 1): We're going to consider a training case in which the ***correct*** answer is ***one***.
* For this kind of training case, the weight vector needs to be on the correct side of the Hyperplane in order to get the answer right.
* It needs to be on the same side of the hyperplane as the direction in which the training vector points.

|  |  |
| --- | --- |
| * For any weight vector like the green one, that's on right side of the hyperplane, the ***angle*** with the ***input*** vector will be **less** than **90** **degrees**. * So, the scaler product of the input vector with a weight vector will be positive. * Since we already got rid of the threshold, that means the perceptron will give an output of **1**. It'll say **yes**, and so we'll get the **right** **answer**. * Conversely, if we have a weight vector like the red one, that's on the wrong side of the plane, the ***angle*** with the ***input*** vector will be ***more*** than ***90 degrees***, so the scalar product of the weight vector and the input vector will be negative. * So, the perceptron will say, ***no*** or ***zero***, and in this case, we'll get the wrong answer. |  |

* So, to summarize:
* On *one side* of the plane, all the weight vectors will get the *right answer*.
* On the *other side* of the plane, all the possible weight vectors will get the *wrong answer*.

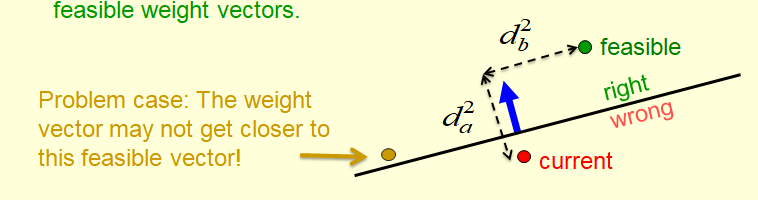
|  |  |
| --- | --- |
| * Case 2 (correct is 0): Let's look at a different training case, in which the ***correct*** answers are ***zero***. We've chosen a different input vector for this input vector, the right answer is zero. * In this case, any weight vectors will make an angle of less than **90** degrees with the *input vector*, will give us a *positive scalar product*, the perceptron outputs yes or one, and it will get the answer wrong. * Conversely the input vector on the other side of the plane, will have an ***angle*** of ***greater*** than ***90*** degrees. And it will correctly give the answer of zero. |  |

|  |  |
| --- | --- |
| * The cone of feasible solutions: To get all *training* cases *right* we need to *find a point* on the *right side* of *all* the *planes*. There may not be any such point! * Let's put those two training cases together in one picture of weight space. * We can see that there's a cone of possible weight vectors. And any weight vectors inside that cone, will get the right answer for both training cases. * Of course, there doesn't have to be any cone like that. * There may not be any weight vectors that get the right answers for all of the training cases. But *if there are any*, they'll *lie in a cone*. * So the learning algorithm needs to consider the training cases (one at a time) in such a way that it eventually lies in this cone of feasible solutions. |  |

* Convex Learning Problem: In general in machine learning if you can get a convex learning problem, that makes life easy.
* If you get any *good weight vector*, that works for all the training cases, it'll lie on this hyper-cone.
* If you take the *average of those two weight vectors* (which works for all the training cases), that will also lie on the cone.
* That means the problem is convex. The average of two solutions is itself a solution.
* If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its ***apex*** at the ***origin***.
* So the average of two good weight vectors is a good weight vector.
* The problem is convex.

**2.4 Why the learning works**

* Why the learning procedure works (first attempt):
* We're going to, look at a proof that the perceptron learning procedure will eventually get the *weights into* the *cone of feasible solutions*.
* We going to use our geometric understanding of what's happening in weight space as perceptrons learns, to get a proof that the perceptron will eventually find a weight vector that it's the right answer for all of the training cases, if any such vector exists.
* Let's assume that *there* is a *vector* that *gets* the *right answer* for *all* *training* *cases*. We'll call that a ***feasible vector***. Shown by the green dot in the *diagram*.
* We start with the weight vector that's getting some of the training cases wrong (shown by the red dot).
* What we want to show, or the idea for the proof is that, ***every time it gets a training case wrong, it will update the current weight vector. In a way that makes it closer to every feasible weight factor.***
* Hopeful Claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors.



* Consider the ***squared distance*** between any *Feasible weight vector* and the *Current weight vector*.
* So we represent the squared distance of the current weight vector from a feasible weight vector, as the sum of a *squared distance along the line of the input vector* that defines the *training case*, and another *squared difference orthogonal to that line*.
* The orthogonal squared distance won't change, and the squared distance along the line of the input vector (i.e. ) will get smaller.
* If you look at the feasible weight vector in gold-color, it's just on the right side of the plane that defines one of the training cases.
* The current weight vector is just on the wrong side, and the input vector is quite big.
* So when we add the input vector to the current weight vector, we actually get further away from that gold feasible weight vector.

So our hopeful claim doesn't work, but we can fix it.

* Why the learning procedure works (attempt 2):
* Now we're gonna define a Generously Feasible Weight Vector. That's a weight vector that not only gets every training case right, but it gets it right by at least a certain margin.
* The margin is as big as the input vector (blue-arrow) for that training case. It creates a feasible region.
* So we take the cone of feasible solutions, and inside that we have another cone of generously feasible solutions.
* Which *get everything right* by *at least* the size of the input vector.

|  |  |
| --- | --- |
| * And now, our proof will work. * Now we can make the claim that: every time the perceptron makes a mistake, the *squared distance* to all of the generously feasible weight vectors would be decreased by at least the *squared length* of the input vector, which is the update we make (update vector). |  |

* Informal sketch of proof of convergence:
* Every time the *preceptron* makes a mistake, the *current weight* vector moves and it *decreases* its *squared distance* from every *generously* *feasible weight* vector (every weight vector in the “generously feasible” region), by at least the *squared length* of the current *input vector*.
* So the squared distance to all the generously feasible weight vectors decreases by at least that squared length of the input vector.
* Assuming that none of the input vectors are infinitesimally small. That means that, after a *finite number of mistakes* the weight vector must lie in the feasible region if this region exists.
* Notice it doesn't have to lie in the generously feasible region, but it has to get into the feasible region to stop it making mistakes.
* Notice, it all depends on the assumption that there is a *generously feasible weight vector*.
* If there is no such vector, the whole proof falls apart.

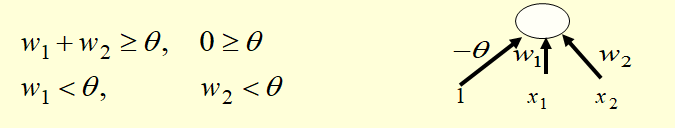
**2.5 What PERCEPTRONS can't do**

Now let's discuss the limitations of perceptrons.

* One limitation is: it depends on the kinds of features you use.
* If you use the right features, you could do *almost anything*.
* If you use the wrong features, they become *extremely limited to do things*.
* It emphasizes that the *difficult bit of learning* is to learn the right features.
* There's still a lot you can do with learning, *even if you do not learn* ***features***.
* For example, if you want to say whether a sentence is a *plausible English sentence*, you could hand define a *huge* number of *features*, and then learn how to *write* them in order to decide whether *particular sentence* is likely a *good English sentence*.

But, in the long run you need to learn features.

* The limitations of Perceptrons:
* The reason that neural network research came to a halt in the late 1960s and early 1970s is that perceptrons were shown to be very limited, and we're now gonna understand what those limitations are.
* If you'd like to choose the features by hand, and if you use enough features, you can make the perceptron do almost anything.
* For binary input vectors, we can have a separate feature unit for each of the exponentially many binary vectors and so we can make any possible discrimination on binary input vectors.
* This type of table look-up won’t generalize.
* Once you've decided the hand coded features, i.e. once they've been determined, there are very strong limitations on what a perceptron can learn to do.
* What binary threshold neurons cannot do:
* A binary threshold output unit cannot even tell if *two single bit features are the same*!
* Positive cases (same): **(1,1) -> 1; (0,0) -> 1**
* Negative cases (different): **(1,0) -> 0; (0,1) -> 0**
* The four input-output pairs give four inequalities that are impossible to satisfy:



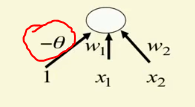
* There's a very simple things that it can't learn to do. The simplest example is consider a problem in which there's two positive cases and two negative cases.
* The features are just single bit features, that have values either one or zero.
* So the two positive cases consist of both features being on and both features being off. And the right answer is **1**.
* The two negative cases are when either one of the feature is off. And the right answer is **zero**.
* So all we're asking the binary threshold unit to do is: decide whether the two features have the same value. And they can't even learn to do that. We can prove that algebraically.
* Above four input/output pairs give us four inequalities, and it's impossible to satisfy them.
* The first positive case , gives us the inequality that:

So that it returns an output 1.

* Then the second positive case gives us the inequality that:
* The negative cases and give us the inequalities that:
* Now if you take those first two inequalities and you add them up, you get:
* And if you take the second two inequalities and you add them up, you get:

So there's clearly no way to satisfy all four inequalities.

* Or to put it another way, if you look at the binary decision unit where we're going to put the threshold as a negative weight on an input line that always has value of **1**.



There's no way to set the threshold in the two weights, so it gets all four cases right.

* A geometric view of what binary threshold neurons cannot do:
* Imagine " data-space" in which the axes correspond to components of an input vector.
* Each input vector is a point in this *space*.
* A weight vector defines a plane in *data-space*.
* The weightplane is perpendicular to the weight vector and *misses* the *origin* by a distance equal to the threshold.
* Data Space: Let's see this geometrically. So we're going to imagine a Data Space now, in which the axis correspond to components of an input vector.
* So in this space each point corresponds to a data point.
* And, a weight vector is going to find a plane in this space.
* We're going to make each point be an input vector and we're going to use a weight vector to define a plane in the Data Space.
* The plane defined by the weight vector is going to be perpendicular to the weight vector and it's going to miss the origin by a distance equal to the threshold.
* So it's just the opposite of what we're doing with weight space. In weight space we made each point be a weight vector, and we used a plane, to define an input case.
* Of course that plane was defined by an input vector.

|  |  |
| --- | --- |
| * In this picture, we can see the four data cases there, and for the two data cases in red, we need to give an output of zero. * with the two data cases in green, we need to put an output of one. * That means we need the green cases to be on the side of the weight plane where the output is one and we need the red cases to be on the side where the output is zero, and we obviously cannot arrange the weight plane so that's true. * Not Linearly Separable Cases: We call a set of cases like that, where there's no hyperplane that will separate the cases where we want the answer to be one, *from the cases* where we want the *answer* to be zero. * We call that a set of training cases that's Not Linearly Separable. |  |

* **Discriminating *simple patterns* under *TRANSLATION* with *Wrap-Around*:**
* *Suppose* we just use ***pixels***as the ***features***.
* Can a ***binary threshold unit*** discriminate between ***different patterns*** that have the ***same number*** of on ***pixels***?
* Not if the ***patterns*** can ***translate*** with ***wrap-around***!
* *Since PERCEPTRONS are much more general, there's more devastating example:* when we try and discriminate *simple patterns* that have to retain the identity when you *translate* them with wrap-around.
* *The idea is that:* we want to recognize a pattern and we want to *recognize it even when it's translated.*
* Suppose we use pixels as the features.
* **The question is:** *can a binary threshold unit* ***discriminate*** *between two* ***different patterns****.*
* We'll call one positive example and the other's negative examples if they've got the same number of pixels in them.
* **The answer is:** No. *It can't discriminate two patterns out of the* ***same number of pixels*** if that discrimination has to work when the patterns are Translated and if the patterns can wrap-around when translate.

|  |  |
| --- | --- |
| * If you look at these examples of pattern A, in a *one-dimensional image*: * Pattern A has four pixels that are on like a little bar code (the first case). * Notice it's the same pattern when we translate it a bit to the right (the second case: move the pattern 2 pixels right). * In the third case, the pattern goes off the ***right hand end***, and comes back on the ***left*** ***hand end***. * So it's the same pattern that's been translated with some wrap-around. * In example of pattern B, it also has four pixels, but in a different arrangement. * Also in the *third case of pattern* ***B***, it's been translated with wrap-around. So that's still an example of pattern B. |  |

* For two sets of patterns like that, a binary threshold unit cannot learn to discriminate them.
* Sketch of a proof that a binary decision unit cannot discriminate patterns with the same number of on pixels (assuming translation with wraparound):
* For ***pattern A***, use training cases in all possible translations.
* Each pixel will be activated by 4 different translations of pattern A.
* So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
* For ***pattern B***, use training cases in all possible translations.
* Each pixel will be activated by 4 different translations of pattern B.
* So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
* But to discriminate correctly, every single case of pattern A must provide more input to the decision unit than every single case of pattern B.
* This is impossible if the sums over cases are the same.
* Consider for the *positive examples*, we have ***pattern A*** in all *possible translations*.
* Now since pattern A has four on pixels, that means if we look at any pixel on the retina, there'll be ***four different positions*** in which we can put ***pattern A*** that will ***activate*** ***that pixel***.
* So ***each pixel*** will be *activated* by *four different translations* of pattern A.
* That means that, the total input received by the decision unit, *over all those various translations of pattern A*, would be four times the *sum of all the weights* in the PERCEPTRON, because each pixel will activate the *decision* unit *four different times*.
* So, summed over all those patterns will get four times the sum of the weights.
* Now consider pattern B. We're going to make the negative cases *be pattern B*, in *all possible translations*. And again, each pixel will be activated by four different translations of pattern B.
* So the *total input* of the *decision unit receives* and, over all those different translations of pattern B, will again be *four times the sum* of all the *weights*.
* But the *PERCEPTRON*, in order to *discriminate correctly*, has to have weights. So that every single case of pattern A provides more input to the *decision unit* than *every single case of pattern B*.
* And that's clearly impossible if you sum of all these cases, all those different versions of pattern A and all of those different versions of pattern B, provide exactly the same amount of input to the decision unit.

So we've proved that a *PERCEPTRON cannot recognize patterns under translation if we allow wrap-around*.

It's a particular case of Minsky and Papert's *group invariance theorem*.

And that result is DEVASTATING for perceptrons, it was *Historically Devastating*.

* Why this result is devastating for Perceptrons:
* The whole point of *pattern recognition* is to *recognize patterns* despite *transformations* like ***translation***.
* Minsky and Papert’s “Group Invariance Theorem” says that the part of a Perceptron that learns cannot learn to do this if the transformations form a GROUP.
* Translations with wrap-around form a GROUP.
* To deal with such transformations, a Perceptron needs to use multiple feature units to recognize transformations of informative sub-patterns.
* So the tricky part of pattern recognition must be solved by the ***hand-coded feature detectors***, not the ***learning procedure***.
* The whole point of pattern recognition is to recognize patterns that undergo *transformations* and see that they're *still the same* *pattern*, despite the transformation (e.g. translation).
* When Minsky and Papert showed that a PERCEPTRON couldn't do that if the *transformations formed a group*, (i.e. the *learning part* of a *perceptron couldn't learn to do that*), it became clear that the claims that have been made for what perceptrons could learn were somewhat exaggerated, and that to get them to do anything interesting.
* You had to choose just the right features to make it fairly easy for the last stage to learn the classification.
* So the translations within our prime form a GROUP and, Minsky and Papert proved a general theorem for *transformations that form a GROUP*, are making it impossible, for the learning part of a perceptron to do the recognition.
* The perceptron architecture can still do the recognition, but you have to organize the features so they do the difficult part.
* So we have to have multiple feature units that recognize informative sub patterns that tell you something about what class it is, and we have to have separate feature units for each position of those informative sub patterns, if we're trying to recognize under translation.
* So the tricky part of pattern recognition has to be solved by the hand-coded feature detectors, not the learning procedure.

The *temporary conclusion* from this is that *perceptrons* are *no good* and therefore *neural networks are no good*.

* Learning with Hidden units: The longer term conclusion is that *neural networks are only gonna be really powerful* *if we can learn the feature detectors*.
* It's not enough just to learn weight some feature detectors, we have to learn the *feature detectors themselves*.
* The ***second generation* of *neural networks***, which we'll come to in the next lecture, was all about *how you learn* the *feature detectors*.
* It took twenty years before people figured out how to do that.
* So, networks without hidden units are very *limited* in what they can learn to model.
* If we add *more layers of linear units*, it doesn't help because *everything is linear*.
* We can make them much more powerful by putting in hand coded hidden units but they're *not really hidden units* because we hand coded them. We told them what to do.
* It's not enough just to have fixed output non-linearity. What we need is multiple layers of adaptive non-linear hidden units. And the problem is how can we train such NN.
* We need a way to adapt all the weights not just the last layer like in a perceptron, and that's hard.
* In particular, leaning the weights go in to the Hidden Units, that's equivalent to learning features.
* And that's the hard thing to do. Because nobody is *telling us directly*, what the *hidden unit should be doing*, when they should be *active* and, when they should *not be active*.
* Real problem is, *how do we figure out how to learn these weights go into hidden units* so that the ***hidden units*** turn into the kinds of ***feature detectors*** we need for solving a problem, when nobody is telling us what the featured detector should be.
* ***Networks without hidden units*** are very limited in the *input-output mappings* they can learn to model.
* More layers of linear units do not help. It's still linear.
* Fixed output non-linearities are not enough.
* We need multiple layers of *adaptive*, non-linear hidden units. But how can we train such nets?
* We need an efficient way of *adapting* all the weights, not just the last layer. This is hard to do.
* *Learning the weights going into hidden units* is equivalent to learning features.
* This is difficult because nobody is telling us directly what the hidden units should do.